

Fast Multiple Kernel Learning With Multiplicative Weight Updates*

John Moeller Parasaran Raman Avishek Saha Suresh Venkatasubramanian[†]

March 8, 2013

Abstract

In this work, we propose a fast algorithm for multiple kernel learning (MKL). Our proposed approach builds on the original QCQP formulation of Lanckriet et al. [12]. It uses a multiplicative weight update based approximation for the underlying SDP, exploiting a careful reformulation of the MKL problem as well as a novel fast matrix exponentiation routine for QCQP constraints that might be of independent interest. Our method avoids the use of commercial nonlinear solvers, and scales efficiently to much larger data sets than most prior methods can handle. Empirical evaluation on eleven datasets shows that our method is significantly faster than most current MKL methods and compares favorably with a uniform unweighted combination of kernels.

1 Introduction

Kernel methods have been extremely successful in machine learning applications [18, 19]. Since the success of these methods rely on an appropriate choice of kernel (or kernels), there has been extensive research on *learning a combination of multiple kernels* for a given task. This approach of learning multiple kernels outperforms [12, 20] algorithms that operate with a single kernel and, as a result, *multiple kernel learning* (MKL) has found successful applications in a wide variety of learning tasks and domains [12, 4, 2, 27, 9, 26, 16, 20].

Support vector machines (SVM) use kernels directly, and a natural approach to multiple kernel learning is to jointly optimize the SVM task and the choice of kernels. This approach, pioneered by Lanckriet et al. [12], exploits convex optimization at the heart of both problems. It is theoretically elegant, but requires repeated invocations of semidefinite solvers which make the algorithm accurate but slow. Followup work focuses on two lines of attack: methods that speed up the solvers by replacing the semidefinite formulation with other optimizations, and methods that bypass the solvers by alternately optimizing the classifier and the kernel combination parameters. The latter approach, while faster, loses the theoretical guarantees of the former. Most importantly, none of these approaches scale effectively beyond a few thousand points.

In this work, we present a theoretically sound and practically efficient MKL algorithm. Based on the original formulation [12] of MKL as a QCQP, we propose a fast MKL algorithm (MWUMKL) that (a) does not require commercial solvers (b) does not make explicit calls to SVMs (unlike alternating optimization based methods), and (c) provably converges to an accurate solution in a fixed number of iterations. Our techniques draw on matrix multiplicative weight update (MWU) based approximation algorithms for SDPs. We make significant modifications to the algorithm to take advantage of the structure of the MKL problem;

*This research is partially supported by the National Science Foundation under grant CCF-0953066.

[†]{moeller,praman,avishek,suresh}@cs.utah.edu

these include a fast routine for *exact* matrix exponentiation, a way to do away with the binary search inherent in the MWU framework, and a number of optimizations to the primal-dual core of the MWU method. A detailed evaluation on eleven datasets shows that our proposed algorithm (a) has comparable accuracy to prior approaches (b) is significantly faster, especially as the data size increases beyond a few thousand points, and (c) compares favorably with the *uniform* heuristic that merely averages all kernels without searching for an optimal combination. As has been recently noted [6], this heuristic is a strong baseline for the evaluation of MKL methods.

Outline. We review the convex formulation of multiple kernel learning in Section 2. Section 3 describes both the MWU framework and our algorithm. In Section 4 we empirically evaluate the quality and scalability of MWUMKL. Finally, in Section 5 we present our observations and highlight future directions.

Related Work. In practice, since the space of all kernels can be unwieldy, many methods operate by fixing a base set of kernels and determining an optimal (conic) combination. Pavlidis et al. [16] proposed assigning equal preference to all kernels and simply using an unweighted sum of kernel functions (UNIFORM). In their seminal work (SDPMKL), Lanckriet et al. [12] proposed to simultaneously train an SVM as well as learn a convex combination of kernel functions. The key contribution was to frame the learning problem as an optimization over positive semi-definite kernel matrices which in turn reduces to *quadratically constrained quadratic programming* (QCQP). Soon after, Bach et al. [4] proposed a block-norm regularization method based on *second order cone programming* (SOCP).

In a bid to improve efficiency, many MKL algorithms started using alternating optimization, alternating between updating the classifier parameters and the kernel weights. For example, Sonnenburg et al. [20] modeled the MKL objective as a cutting plane problem and solved for kernel weights using Semi-Infinite Linear Programming (SILP) techniques. Rakotomamonjy et al. [17] used sub-gradient-descent-based methods to solve the MKL problem (SIMPLEMKL). An improved level set based method (LEVELMKL) that combines cutting plane models with projection to level sets was proposed by Xu et al. [24]. More recently Xu et al. [25] derive a variant of the equivalence between group lasso and the MKL formulation that leads to closed-form updates for kernel weights (GROUPMKL).

Other works in MKL literature study the use of different kernel families, such as Gaussian families [13], hyperkernels [14] and nonlinear families [22, 7]. Kloft et al. [10, 11] and Vishwanathan et al. [23] introduce regularization based on the ℓ_p norm. Cortes et al. [8] propose a two-stage approach to optimize the kernel weights in the first stage and learn a standard SVM in the second stage. In addition, Orabona and Luo [15] study online algorithms for MKL based on stochastic gradient descent.

2 Background

Let $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) \in \mathbb{R}^{n \times d}$ denote a collection of n training samples in a d -dimensional space. Let $\mathbf{y} = (y_1, y_2, \dots, y_n) \in \{-1, +1\}^n$ denote the binary class labels for the data points in \mathbf{X} . The problem of multiple kernel learning can be cast as the problem of minimizing $\frac{1}{2} \|\mathbf{w}\|^2$ such that $y_j(\langle \mathbf{w}, \Phi(\mathbf{x}_j) \rangle + b) \geq 1$ for all j where Φ is the feature map associated with the (unknown) kernel κ . With the constraint that κ can be written as $\kappa = \sum_{i=1}^m \mu_i \kappa_i$ (with a trace regularizer) the dual problem takes the following form [12]

$$\begin{aligned} & \min_{\mathbf{K}} \max_{\alpha} \quad 2\alpha^\top \mathbf{1} - \alpha^\top \mathbf{G} \alpha \\ \text{s.t.} \quad & \mathbf{K} = \sum_{i=1}^m \mu_i \mathbf{K}_i, \quad \text{tr}(\mathbf{K}) = c, \quad \mathbf{K} \succeq 0, \quad \mu \geq 0 \end{aligned}$$

where \mathbf{K}_i is the Gram matrix for \mathbf{X} associated with the kernel function $\kappa_i(\cdot, \cdot)$, $\mathbf{G}_i = \text{diag}(\mathbf{y}) \mathbf{K}_i \text{diag}(\mathbf{y})$ and $\mathbf{G} = \sum_{i=1}^m \mu_i \mathbf{G}_i$. The kernel \mathbf{K} optimizing this expression can be found by solving the following convex optimization problem [12, Theorem 20]:

$$\begin{aligned} & \max_{\alpha, s} \quad 2\alpha^\top \mathbf{1} - cs \\ \text{s.t.} \quad & s \geq \frac{1}{r_i} \alpha^\top \mathbf{G}_i \alpha, \quad \alpha^\top \mathbf{y} = 0, \quad \alpha \geq 0 \end{aligned} \tag{2.1}$$

where $r \in \mathbb{R}^m$ and $\text{tr}(\mathbf{K}_i) = r_i$. This program is an example of a class of convex programs called *quadratically constrained quadratic programs* (QCQPs). Lanckriet et al. [12] propose solving this program with a *second order cone program* (SOCP) solver such as Mosek [1] or SeDuMi [21], as this type of solver is very efficient and the class of QCQPs is a subset of SOCPs.

We note that (2.1) is the *hard-margin* version of the MKL problem: there are soft-margin variants that are also amenable to our methods [12]. For the 1-norm soft margin, an additional constraint is added, namely that all the terms of α are bounded from above by the margin constant C . For the 2-norm soft margin, another term $\frac{1}{C} \alpha^\top \alpha$ appears in the objective, or we can simply add a constant multiple of the objective to each \mathbf{G}_i .

Notation. We will denote vectors by boldface lower case letters like \mathbf{z} , and matrices by bold uppercase letters \mathbf{M} . The operator \succeq is used to denote positive semi-definiteness; $\mathbf{M} \succeq 0$ states that \mathbf{M} is positive semidefinite. The matrix inner product $\mathbf{A} \bullet \mathbf{B} = \text{Tr}(\mathbf{AB}) = \sum_{i,j} A_{ij} B_{ij}$. We denote a zero matrix $\mathbf{0}$ (in boldface, like vectors and matrices), and an all-ones matrix $\mathbf{1}$.

3 Algorithm

The central idea of this paper is to solve the multiple kernel learning problem as formulated in (2.1) via the matrix multiplicative weight update method (MWU), specifically the approach for solving semidefinite programs (SDP) employed by Arora and Kale [3]. This method is a general template for solving such programs and requires a significant amount of customization and optimization for each specific instance. In what follows, we will describe how to apply it to the MKL problem, and highlight places where the structure of our problem allows us to optimize beyond what the original template allows.

An SDP reformulation. The first step is to reformulate the QCQP in (2.1) as an SDP. While this is trivial in general (every QCQP is an SDP) we require a specific form of the SDP in order to apply the MWU. This form is given as

$$\min_{\mathbf{z}} \quad \mathbf{g} \cdot \mathbf{z} \quad \text{s.t.} \quad \sum_j \mathbf{F}_j z_j \succeq \mathbf{H} \quad \mathbf{z} \geq 0. \tag{3.1}$$

Let $\mathbf{A}_i^\top \mathbf{A}_i = \frac{c}{r_i} \mathbf{G}_i$ for all $i \in [0..m]$. Since each $\mathbf{G}_i \succeq 0$ and $c \geq 0$, all the \mathbf{A}_i exist. Algebraic manipulation of (2.1) allows us to express it in the form of (3.1), with $\mathbf{H} = \begin{pmatrix} \mathbf{I}_n & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}$, $\mathbf{z} = (\alpha, s)$, $\mathbf{F}_j = \begin{pmatrix} \mathbf{0} & \mathbf{A}_i(j) \\ \mathbf{A}_i(j)^\top & 0 \end{pmatrix}$

and $F_{n+1} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0}^\top & c \end{pmatrix}$, where $\mathbf{A}_i(j)$ is the j^{th} column of \mathbf{A}_i . A more convenient formulation combines the matrix terms in the semidefinite constraint into a set of matrix constraints Q_i , yielding the form

$$\begin{aligned} \min_{\alpha, s} \quad & cs - 2\alpha^\top \mathbf{1} \\ \text{s.t.} \quad & \mathbf{Q}_i(\alpha) = \begin{pmatrix} \mathbf{I}_n & \mathbf{A}_i \alpha \\ (\mathbf{A}_i \alpha)^\top & cs \end{pmatrix} \succeq \mathbf{0} \quad \forall i \in [1..m], \quad \alpha^\top \mathbf{y} = 0, \quad \alpha \geq \mathbf{0}, \end{aligned} \quad (3.2)$$

Algorithm 1 MWU template [3]

Given primal variables $\mathbf{L}^{(1)} = (1/n)\mathbf{I}$, error ε , # of iterations T , and guess ω

```

for  $t = 1 \dots \tau$  do
  Run ORACLE with  $\mathbf{L}^{(t)}$ 
  if Oracle fails then
    Return and increase  $\omega$ 
  else
     $\mathbf{z}^{(t)} \leftarrow$  dual variable obtained by ORACLE
     $\mathbf{M}^{(t)} \leftarrow (\sum_j \mathbf{F}_j \mathbf{z}_j^{(t)} - \mathbf{H} + \rho \mathbf{I}) / 2\rho$ 
     $\mathbf{W}^{(t+1)} \leftarrow (1 - \varepsilon) \sum_{i=1}^t \mathbf{M}^{(i)}$ 
     $\mathbf{L}^{(t+1)} = \frac{\mathbf{W}^{(t+1)}}{\text{Tr} \mathbf{W}^{(t+1)}}$ 
  Return  $\mathbf{L}^\top$  and decrease  $\omega$ 

```

The MWU framework. At a high level, the MWU is a primal-dual scheme that works as follows. It starts with a guess ω for the optimal value of the SDP. Assuming that this guess for the optimal value is correct, the program reduces to a feasibility problem in the primal and dual variables. The algorithm then attempts to find either a feasible primal or feasible dual assignment such that this guess is achieved.

The search for feasible values is conducted by an exponential re-weighting search. The process starts with some (typically infeasible) primal solution. In each step, an oracle (which solves a simple linear program) is used to determine a candidate dual \mathbf{z} . If this is possible, then the dual values are used to guide the search for a new (less infeasible) primal solution by exponentially re-weighting primal constraints that are more important, and the process repeats.

If after a specific number of iterations the process completes, then we have found a solution for the current guess ω and we can try reducing it. If at any step the process fails (because the oracle cannot find the desired \mathbf{z}), then the guess was too low and is increased. This binary search over ω eventually yields a solution guaranteed to be within a $(1 + \varepsilon)$ factor of the optimal solution. The inner routine (inside the binary search) is summarized in Algorithm 1 (ρ is dependent upon ORACLE).

In order to make the framework feasible and efficient for solving MKL, we need to solve a number of problems. In what follows we describe these one by one, and then present the final algorithm.

3.1 Eliminating Binary Search

The first optimization we can perform is to note that *we can circumvent the binary search on ω entirely* and run Algorithm 1 only once.

Since every iteration of the binary search is reduced to a feasibility problem we can restate (2.1) as:

$$\omega = (cs - 2\alpha^\top \mathbf{1}) \quad \text{s.t.} \quad s \geq \frac{1}{r_i} \alpha^\top \mathbf{G}_i \alpha, \quad \alpha^\top \mathbf{y} = 0, \quad \alpha \geq 0.$$

We can further transform the problem by taking advantage of the KKT conditions. The support constraints of the SVM problem can be written as $\mathbf{G}\alpha + b\mathbf{y} \geq \mathbf{1}$. If we multiply both sides of this inequality by α^\top then it becomes an equality (by complementary slackness): $\alpha^\top \mathbf{G}\alpha = \alpha^\top \mathbf{1}$. cs is a substitution for $\alpha^\top \mathbf{G}\alpha$ in the MKL problem [12], so $cs = \alpha^\top \mathbf{1}$ as well. The objective function is *linear*, so we can scale s and α with any positive factor that we like, and use the fact that $cs = \alpha^\top \mathbf{1} = |\omega|$ to transform the problem:

$$\begin{aligned} &\text{find} \quad \hat{\alpha} \\ &\text{s.t.} \quad 1/(c|\omega|) \geq \frac{1}{r_i} \hat{\alpha}^\top \mathbf{G}_i \hat{\alpha}, \quad \hat{\alpha}^\top \mathbf{y} = 0, \quad \hat{\alpha}^\top \mathbf{1} = 1, \quad \hat{\alpha} \geq 0, \end{aligned}$$

where $\alpha = |\omega| \hat{\alpha}$. The first constraint can be transformed back into an optimization; that is,

$$\min_{\omega} \max_{\hat{\alpha}, i} \frac{1}{r_i} \hat{\alpha}^\top \mathbf{G}_i \hat{\alpha},$$

subject to the remaining linear constraints. Because ω does not figure into the maximization, we can compute ω simply by maximizing $\frac{1}{r_i} \hat{\alpha}^\top \mathbf{G}_i \hat{\alpha}$. Practically, this means that we simply add the constraint $\alpha^\top \mathbf{1} = 1$, and the “guess” for ω is set to -1 . We then *know* the objective, and only one iteration is needed, so the binary search is eliminated.

3.2 Matrix Exponentiation

A critical step in the MWU process is computing $(1 - \varepsilon)^{\sum_{i=1}^{\tau} \mathbf{M}^{(i)}}$ (see Algorithm 1). For MKL, $\mathbf{M}^{(t)}$ is a block-diagonal matrix, with m blocks of the form $\mathbf{M}_i^{(t)} = \frac{1}{2\rho} (\mathbf{Q}_i(\alpha^{(t)}) + \rho \mathbf{I}_{n+1})$. The matrix $(1 - \varepsilon)^{\sum_{i=1}^{\tau} \mathbf{M}^{(i)}}$ will also be block-diagonal, with m blocks $(1 - \varepsilon)^{\sum_{i=1}^{\tau} \mathbf{M}_i^{(i)}}$. Therefore we just need a way to exponentiate $\sum_{i=1}^{\tau} \mathbf{M}_i^{(i)} = \frac{1}{2\rho} (\mathbf{Q}_i(\sum_{i=1}^{\tau} \alpha^{(i)}) + \rho \mathbf{I}_{n+1})$ for each i .

In general, matrix exponentiation is expensive, and Arora and Kale [3] suggest ways to approximate this computation. However, in our case, the particular form of \mathbf{Q}_i (c.f. (3.2)) yields a more elegant closed form solution:

Lemma 3.1. *The exponential of a matrix in the form $\begin{pmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & b \end{pmatrix}$, where a and b are nonnegative, is*

$$e^\phi \begin{pmatrix} (\cosh \psi + \sinh \psi \cos \gamma) \hat{\mathbf{u}} \hat{\mathbf{u}}^\top & \sinh \psi \sin \gamma \hat{\mathbf{u}} \\ \sinh \psi \sin \gamma \hat{\mathbf{u}}^\top & \cosh \psi - \sinh \psi \cos \gamma \end{pmatrix} + e^a \begin{pmatrix} \mathbf{I}_n - \hat{\mathbf{u}} \hat{\mathbf{u}}^\top & 0 \\ 0 & 0 \end{pmatrix}, \quad (3.3)$$

where $\hat{\mathbf{u}}$ is the unit vector $\mathbf{u}/\|\mathbf{u}\|$, $\phi = (a+b)/2$, $\psi = \sqrt{(a-b)^2/4 + \|\mathbf{u}\|^2}$, and $\gamma = \tan^{-1}(2\|\mathbf{u}\|/(a-b))$.

Proof. We symbolically exponentiate an $n+1 \times n+1$ matrix of the form

$$M = \begin{pmatrix} aI & \mathbf{u} \\ \mathbf{u}^\top & b \end{pmatrix}.$$

Since this matrix is real and symmetric, its eigenvalues λ_i are positive and its unit eigenvectors \mathbf{v}_i form an orthonormal basis. The method that we use to symbolically exponentiate it is to express it in the form

$$M = \sum_{i=1}^n \lambda_i \mathbf{v}_i \mathbf{v}_i^T.$$

The exponential then becomes

$$e^M = \sum_{i=1}^n e^{\lambda_i} \mathbf{v}_i \mathbf{v}_i^T.$$

As a matter of notation, let $\hat{\mathbf{u}}$ be the unit vector such that $\|\mathbf{u}\| \hat{\mathbf{u}} = \mathbf{u}$.

Eigenvalues. The characteristic equation for M is not difficult to calculate. It is:

$$(\lambda - a)^{n-2} (\lambda^2 - (a+b)\lambda + ab - \|\mathbf{u}\|^2). \quad (3.4)$$

This yields $n-2$ eigenvalues equal to a , and the other two equal to $(a+b)/2 + \sqrt{(a-b)^2/4 + \|\mathbf{u}\|^2}$ and $(a+b)/2 - \sqrt{(a-b)^2/4 + \|\mathbf{u}\|^2}$. We label them λ_1 and λ_2 , respectively, and the rest are equal to a .

Eigenvectors. To compute the eigenvectors of M , we compute the solution to $(\lambda_i I - M)\mathbf{v}_i$.

For λ_1 , we have

$$\begin{pmatrix} (\lambda_1 - a)I & -\mathbf{u} \\ -\mathbf{u}^T & \lambda_1 - b \end{pmatrix} \begin{pmatrix} \mathbf{v}'_1 \\ 1 \end{pmatrix} = \mathbf{0}$$

(we will scale \mathbf{v}_1 later). This yields $\mathbf{v}'_1 = \mathbf{u}/(\lambda_1 - a)$. We want to normalize this, so we compute

$$\begin{aligned} \|\mathbf{v}'_1\|^2 + 1 &= \frac{\|\mathbf{u}\|^2}{(\lambda_1 - a)^2} + 1 = \frac{\|\mathbf{u}\|^2 + (\lambda_1 - a)^2}{(\lambda_1 - a)^2} \\ &= \frac{(\lambda_1 - a)(\lambda_1 - b) + (\lambda_1 - a)^2}{(\lambda_1 - a)^2} \end{aligned} \quad (3.5)$$

$$\begin{aligned} &= \frac{(\lambda_1 - b) + (\lambda_1 - a)}{\lambda_1 - a} = \frac{2\lambda_1 - (a+b)}{\lambda_1 - a} \\ &= \frac{2\lambda_1 - (\lambda_1 + \lambda_2)}{\lambda_1 - a} = \frac{\lambda_1 - \lambda_2}{\lambda_1 - a}, \end{aligned} \quad (3.6)$$

where (3.5) results from λ_1 being a root of (3.4), and (3.6) is because $a+b = \lambda_1 + \lambda_2$. So \mathbf{v}_1 becomes

$$\begin{aligned} \frac{(\mathbf{v}'_1, 1)}{\|(\mathbf{v}'_1, 1)\|} &= \left(\frac{\mathbf{u}}{\lambda_1 - a}, 1 \right) \sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}} \\ &= \left(\frac{\mathbf{u}}{\sqrt{\lambda_1 - a}\sqrt{\lambda_1 - \lambda_2}}, \sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}} \right) \\ &= \left(\frac{\sqrt{\lambda_1 - a}\sqrt{\lambda_1 - b}}{\sqrt{\lambda_1 - a}\sqrt{\lambda_1 - \lambda_2}} \hat{\mathbf{u}}, \sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}} \right) \\ &= \left(\sqrt{\frac{\lambda_1 - b}{\lambda_1 - \lambda_2}} \hat{\mathbf{u}}, \sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}} \right) \end{aligned}$$

The formula for \mathbf{v}_2 is similar, except with λ_1 and λ_2 reversed, and a sign reversal on \mathbf{v}'_2 :

$$\frac{(\mathbf{v}'_2, 1)}{\|(\mathbf{v}'_2, 1)\|} = \left(-\sqrt{\frac{\lambda_2 - b}{\lambda_2 - \lambda_1}} \hat{\mathbf{u}}, \sqrt{\frac{\lambda_2 - a}{\lambda_2 - \lambda_1}} \right).$$

We can quickly observe, however, that since $\lambda_2 - b = a - \lambda_1$ and $\lambda_2 - a = b - \lambda_1$,

$$\frac{(\mathbf{v}'_2, 1)}{\|(\mathbf{v}'_2, 1)\|} = \left(-\sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}} \hat{\mathbf{u}}, \sqrt{\frac{\lambda_1 - b}{\lambda_1 - \lambda_2}} \right).$$

We let $\alpha = \sqrt{\frac{\lambda_1 - b}{\lambda_1 - \lambda_2}}$ and $\beta = \sqrt{\frac{\lambda_1 - a}{\lambda_1 - \lambda_2}}$ (noting that $\alpha^2 + \beta^2 = 1$), so the unit eigenvectors become

$$\mathbf{v}_1 = (\alpha \hat{\mathbf{u}}, \beta) \quad \mathbf{v}_2 = (-\beta \hat{\mathbf{u}}, \alpha)$$

As for the other eigenvectors, we only care about $\sum_{i=3}^n \mathbf{v}_i \mathbf{v}_i^T$, since all other eigenvalues are equal. But this is just

$$\begin{aligned} \sum_{i=3}^n \mathbf{v}_i \mathbf{v}_i^T &= \sum_{i=1}^n \mathbf{v}_i \mathbf{v}_i^T - \mathbf{v}_1 \mathbf{v}_1^T - \mathbf{v}_2 \mathbf{v}_2^T = I_{n+1} - \mathbf{v}_1 \mathbf{v}_1^T - \mathbf{v}_2 \mathbf{v}_2^T \\ &= I_{n+1} - \begin{pmatrix} \alpha^2 \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \alpha \beta \hat{\mathbf{u}} \\ \alpha \beta \hat{\mathbf{u}}^T & \beta^2 \end{pmatrix} - \begin{pmatrix} \beta^2 \hat{\mathbf{u}} \hat{\mathbf{u}}^T & -\alpha \beta \hat{\mathbf{u}} \\ -\alpha \beta \hat{\mathbf{u}}^T & \alpha^2 \end{pmatrix} \\ &= I_{n+1} - \begin{pmatrix} \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \mathbf{0} \\ \mathbf{0}^T & 1 \end{pmatrix} = \begin{pmatrix} I_n - \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \mathbf{0} \\ \mathbf{0}^T & 0 \end{pmatrix}. \end{aligned}$$

The Exponential. All that remains is to put everything together:

$$\begin{aligned} e^M &= \sum_{i=1}^n e^{\lambda_i} \mathbf{v}_i \mathbf{v}_i^T \\ &= e^{\lambda_1} \begin{pmatrix} \alpha^2 \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \alpha \beta \hat{\mathbf{u}} \\ \alpha \beta \hat{\mathbf{u}}^T & \beta^2 \end{pmatrix} + e^{\lambda_2} \begin{pmatrix} \beta^2 \hat{\mathbf{u}} \hat{\mathbf{u}}^T & -\alpha \beta \hat{\mathbf{u}} \\ -\alpha \beta \hat{\mathbf{u}}^T & \alpha^2 \end{pmatrix} + e^a \begin{pmatrix} I_n - \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \mathbf{0} \\ \mathbf{0}^T & 0 \end{pmatrix}. \end{aligned}$$

Some variable substitutions will give us the form in (3.3); $\lambda_1 = \phi + \psi$, $\lambda_2 = \phi - \psi$, and $\alpha = \cos(\gamma/2)$:

$$= e^\phi \begin{pmatrix} (\cosh \psi + \sinh \psi \cos \gamma) \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \sinh \psi \sin \gamma \hat{\mathbf{u}} \\ \sinh \psi \sin \gamma \hat{\mathbf{u}}^T & \cosh \psi - \sinh \psi \cos \gamma \end{pmatrix} + e^a \begin{pmatrix} I_n - \hat{\mathbf{u}} \hat{\mathbf{u}}^T & \mathbf{0} \\ \mathbf{0} & 0 \end{pmatrix}.$$

□

In each iteration, the matrix to be exponentiated is a sum of matrices of the form $\frac{1}{2\rho}(\mathbf{Q}_i(\sum_{t=1}^{\tau} \alpha^{(t)}) + \rho t \mathbf{I}_{n+1})$, and so Lemma 3.1 can be applied. While the derivation of the exponential is straightforward, practical consideration needs to be given to our implementation of the exponential.

In Lemma 3.1, note that if we give large inputs to the functions \exp , \cosh , and \sinh we will rapidly overflow even a double-precision range. Fortunately $\exp(x)/2$ gets exponentially close to both $\sinh(x)$ and $\cosh(x)$, so above a certain value, we can use \exp alone and throw in a “quashing” factor ($e^{-\phi-q}$) on the result, because it will be normalized out later in the computation.

We can also simplify the exponentiation by observing that $|\omega| = 1$. This makes $\phi = a_{\mathbf{M}} = b_{\mathbf{M}}$, and $\nu = 0$, which then makes ψ_i proportional to $\|\mathbf{u}_i\|$, $\cos(\gamma_i) = 0$, and $\sin(\gamma_i) = \pm 1$ (in our case this is -1). Algorithm 2 contains the pseudocode for the exponentiation.

Algorithm 2 EXPONENTIATE- M

Require: $\mathbf{y}, \alpha, \{\mathbf{G}_i\}, \varepsilon', \rho, t$
 $\phi \leftarrow -\frac{\varepsilon'}{2\rho}(1+\rho)t$
for $i \in [1..m]$ **do**
 $\|\mathbf{u}_i\| \leftarrow \sqrt{\alpha^T \mathbf{G}_i \alpha}$
 $\mathbf{g}_i \leftarrow \frac{1}{\|\mathbf{u}_i\|} \mathbf{G}_i \alpha$
 $\psi_i \leftarrow \frac{\varepsilon'}{2\rho} \|\mathbf{u}_i\|$
 $q \leftarrow \max_i \psi_i$
if $q < 20$ **then**
 for $i \in [1..m]$ **do**
 $l_i^{11} \leftarrow \cosh(\psi_i)$
 $l_i^{12} \leftarrow -\sinh(\psi_i)$
 $e_M \leftarrow 1$
else
 for $i \in [1..m]$ **do**
 $l_i^{11} \leftarrow e^{\psi_i - q}$
 $l_i^{12} \leftarrow -l_i^{11}$
 $e_M \leftarrow 2e^{-q}$ {This is effectively 0}
 $S \leftarrow m(n-1)e_M + 2\sum_i l_i^{11}$
for $i \in [1..m]$ **do**
 $l_i^{11} \leftarrow l_i^{11}/S$
 $l_i^{12} \leftarrow l_i^{12}/S$
 $\mathbf{g} \leftarrow \sum_i 2l_i^{12} \mathbf{g}_i$
return l_i^{12}, \mathbf{g}

3.3 Computing The Oracle

We now need an implementation for the ORACLE routine. For an SDP given by (3.1) with current primal “guess” \mathbf{L} , the oracle returns a $\mathbf{z} \geq 0$ such that $\sum_j (\mathbf{F}_j \bullet \mathbf{L}) z_j \geq \mathbf{H} \bullet \mathbf{L}$. Translating this to the context of MKL, we need an oracle that given \mathbf{L} expressed as a block diagonal matrix with blocks \mathbf{L}_i , finds α such that¹ $\sum_i Q_i(\alpha) \bullet \mathbf{L}_i \geq 0, \alpha \geq 0, \alpha^\top \mathbf{y} = 0, \alpha^\top \mathbf{1} = 1$.

Note that \mathbf{L} comes from the process of exponentiation, and so has special structure that Lemma 3.1 allows us to exploit. Recall that $\mathbf{u}_i = \mathbf{A}_i \alpha$ and $\hat{\mathbf{u}}_i = \mathbf{u}_i / \|\mathbf{u}_i\|$. Let us set

$$l_i^{11} = e^\phi (\cosh \psi + \sinh \psi \cos \gamma), l_i^{12} = e^\phi (\sinh \psi \sin \gamma), l_i^{22} = e^\phi (\cosh \psi - \sinh \psi \cos \gamma)$$

With algebraic manipulation, and the observations that $\hat{\mathbf{u}}_i \hat{\mathbf{u}}_i^\top$ has unit trace and that a_i is the same for all i , $\sum_j (\mathbf{F}_j \bullet \mathbf{L}) z_j \geq \mathbf{H} \bullet \mathbf{L}$ becomes:

$$\left(\sum_{i=0}^m (2l_i^{12} \hat{\mathbf{u}}_i^\top \mathbf{A}_i) \right) \alpha \geq -m(n-1)e^a - \sum_{i=0}^m (l_i^{11} + l_i^{22} cs). \quad (3.7)$$

Since $cs = |\omega| = 1$ from Section 3.1, the right-hand side of this inequality is simply the negative of the trace

¹The MKL formulation introduces additional linear constraints on α that are not strictly part of the original SDP. It is easy to show however that these can be folded into the oracle computation.

of \mathbf{L} , so this simplifies to

$$\left(\sum_{i=1}^m (2\ell_i^{12} \hat{\mathbf{u}}_i^\top \mathbf{A}_i) \right) \alpha^{(t)} \geq -1, \quad (3.8)$$

since \mathbf{L} is normalized with trace 1 in the MWU template.

Algorithm 3 ORACLE-MKL

Require: \mathbf{y}, \mathbf{g}

Ensure: $\alpha \geq 0, \alpha^\top \mathbf{1} = 1, \alpha^\top \mathbf{y} = 0$

$P \leftarrow \{i \mid \mathbf{y}_i = 1\}, N \leftarrow \{i \mid \mathbf{y}_i = -1\}$

$i_P \leftarrow \arg \max_{i \in P} \mathbf{g}_i, i_N \leftarrow \arg \max_{i \in N} \mathbf{g}_i$

$\alpha \leftarrow \mathbf{0}$

$\alpha_{i_P} \leftarrow 1/2, \alpha_{i_N} \leftarrow 1/2$

if α satisfies inequality $(\sum_i 2\ell_i^{12} \mathbf{g}_i)^\top \alpha \geq -1$ **then**

 return α

else

 return FAIL

Satisfying Equation (3.8) We have reduced the oracle problem to finding an α that satisfies Equation (3.8) in addition to the other (linear) constraints on α . Further simplifications make this equation even more tractable. Firstly, observe that $\hat{\mathbf{u}}_i^\top \mathbf{A}_i$ can be rewritten as $\mathbf{g}_i^\top = (c/r_i)^{1/2} \alpha^\top \mathbf{G}_i / (\alpha^\top \mathbf{G}_i \alpha)^{1/2}$, allowing us to simplify (3.8) to $(\sum_i 2\ell_i^{12} \mathbf{g}_i)^\top \alpha \geq -1$.

Note that the coefficient vector $\mathbf{g} = \sum_i 2\ell_i^{12} \mathbf{g}_i$ can be easily computed when we exponentiate the matrix $\mathbf{M}^{(t)}$. In order to find a α that satisfies (3.8), we simply choose the highest elements of \mathbf{g} that correspond to both positive and negative labels, then set each corresponding entry in α to 1/2. If this choice of α does not satisfy (3.8), then we know that the oracle must fail.

Algorithm 3 describes the pseudo-code of ORACLE for MKL.

Notes. The oracle procedure produces a very sparse update to α : in each iteration, only two coordinates of α are updated. This means that each iteration is very efficient, taking only linear time. Further, the trick of replacing \mathbf{u}_i by \mathbf{g}_i means that we never need to explicitly compute \mathbf{A}_i , which in turn means that we do not need to compute the (expensive) square root of \mathbf{G}_i explicitly. Another beneficial feature of the ORACLE for MKL is that terms involving the primal variables \mathbf{L} are either normalized or eliminated, *which means that we never have to explicitly maintain \mathbf{L} .*

3.4 Extracting the solution from the MWU

To extract the kernel weights, we start by observing that $\sum_{i=1}^m \mathbf{Q}_i \bullet \mathbf{L}_i = 0$, by complementary slackness. Performing similar algebra as in Section 3.3, we see that $2 \sum_{i=1}^m \left(\frac{c}{r_i} \alpha^\top \mathbf{G}_i \alpha \right)^{1/2} \ell_i^{12} = 1$. We transform this further to

$$\sum_{i=1}^m \alpha^\top \mathbf{G}_i \alpha (2\ell_i^{12}) \left(\frac{(c/r_i)}{\alpha^\top \mathbf{G}_i \alpha} \right)^{1/2} = 1.$$

This form suggests that $2\ell_i^{12} \left(\frac{(c/r_i)}{\alpha^\top \mathbf{G}_i \alpha} \right)^{1/2}$ is the appropriate choice for μ_i . Indeed, since $\sum_{i=1}^m \mu_i \alpha^\top \mathbf{G}_i \alpha$ is $\alpha^\top \mathbf{G} \alpha$, and because of the KKT conditions of the original problem, we know that $\alpha^\top \mathbf{G} \alpha = \alpha^\top \mathbf{1} = |\omega| = 1$.

3.5 Putting it all together

Algorithm 4 MWU-MKL

Require: $\mathbf{g}^{(1)} = \mathbf{0}$;
 ρ , the width of ORACLE;
 ε , the desired approximation error
Set $\varepsilon' = -\ln(1 - \frac{\varepsilon}{2\rho})$
Set $\tau = \frac{8\rho^2}{\varepsilon^2} \ln(n)$
repeat $\{\tau \text{ times}\}$
 Set $\alpha^{(t)} = \text{ORACLE}(\mathbf{y}, \mathbf{g}^{(t)})$
 Update $\alpha = \alpha + \alpha^{(t)}$
 if ORACLE failed **then**
 return
 Set $\mathbf{M}_i^{(t)} = \frac{1}{2\rho} (\mathbf{Q}_i(\alpha^{(t)}) + \rho \mathbf{I}_{n+1})$
 Set $\mathbf{W}_i^{(t)} = e^{-\varepsilon' \sum_{i=1}^t \mathbf{M}_i^{(t)}} \{\text{Call EXPONENTIATE-}M(\mathbf{y}, \alpha, \{\mathbf{G}_i\}, \varepsilon', \rho, t), \text{Algorithm 2}\}$
 Set $\mathbf{L}_i^{(t+1)} = \mathbf{W}_i^{(t)} / \text{Tr}(\mathbf{W}_i^{(t)})$
 Compute $\mathbf{g}^{(t+1)}$ from $\mathbf{L}^{(t+1)}$, $\{\mathbf{G}_i\}$, and α
until $t = \tau$
return $\frac{1}{\tau} \alpha, \mathbf{L}^{(t+1)}$

Algorithm 4 summarizes the discussion in this section. The parameter ε is the error in approximating the objective function, but its connection to classification accuracy is loose. We set the actual value of ε via cross-validation (see Section 4). The parameter ρ is the *width* of ORACLE, a parameter that indicates how much the solution can vary at each step. ρ is equal to the maximum absolute value of the eigenvalues of $\mathbf{Q}_i(\alpha^{(t)})$, for any i [3].

Running time. Starting innermost, every iteration of Algorithm 4 will require a call to ORACLE-MKL, a call to EXPONENTIATE- M and an update to $\mathbf{G}_i \alpha$ and $\alpha^\top \mathbf{G}_i \alpha$. ORACLE requires a linear search for two maxima in \mathbf{g} , so the first is $O(n)$. The latter are each $O(mn)$, which dominate ORACLE. Algorithm 4 requires a total of τ iterations at most, where $\tau = \frac{8\rho^2}{\varepsilon^2} \ln(n)$. The parameter ρ^2 can be bounded as $O(c)$:

Lemma 3.2. ρ is bounded by $1 + \sqrt{c}/2$.

Proof. ρ is defined as the maximum of $\|\mathbf{Q}(\alpha^{(t)})\|$ for all t . Here $\|\cdot\|$ denotes the largest eigenvalue in absolute value [3]. Because $cs = |\omega| = 1$, the eigenvalues of $\mathbf{Q}_i(\alpha^{(t)})$ are 1 (with multiplicity $n-1$), and $1 \pm \|\mathbf{A}_i \alpha^{(t)}\|$. The greater of these in absolute value is clearly $1 + \|\mathbf{A}_i \alpha^{(t)}\|$. $\|\mathbf{A}_i \alpha^{(t)}\|$ is equal to

$$((\alpha^{(t)})^T \mathbf{A}_i^T \mathbf{A}_i \alpha^{(t)})^{\frac{1}{2}} = \left(\frac{c}{r_i} (\alpha^{(t)})^T \mathbf{G}_i \alpha^{(t)} \right)^{\frac{1}{2}}.$$

$\alpha^{(t)}$ always has two nonzero elements, and they are equal to $1/2$. They also correspond to values of \mathbf{y} with opposite signs, so if j and k are the coordinates in question, $(\alpha^{(t)})^T \mathbf{G}_i \alpha^{(t)} \leq (1/4)(\mathbf{G}_{i(jj)} + \mathbf{G}_{i(kk)})$, because $\mathbf{G}_{i(jk)}$ and $\mathbf{G}_{i(kj)}$ are both negative. Because of the factor of c/r_i , and because r_i is the trace of \mathbf{G}_i , $\|\mathbf{A}_i \alpha^{(t)}\| \leq \sqrt{c}/2$. This is true for any of the i , so the maximum eigenvalue of $\mathbf{Q}(\alpha^{(t)})$ in absolute value is bounded by $1 + \sqrt{c}/2$. \square

Since we only require one run of the main algorithm, the running time is bounded by $O(cmn \ln(n) \frac{1}{\varepsilon^2})$.

4 Experiments

In this section we compare the empirical performance of MWUMKL with other multiple kernel learning algorithms. Our results have two components: (a) *qualitative* results that compares test accuracies on small scale datasets, and (b) *scalability* results that compares training time on larger datasets.

We compare MWUMKL with the following baselines: (a) UNIFORM (un-weighted combination of kernels), (b) SDPMKL [12], (c) SIMPLEMKL [17], (d) LEVELMKL [24], (e) GROUPEMKL [25]. Another related method is SMOMKL [23]: we discuss this in Section 5. We evaluate these MKL methods on binary datasets from UCI data repository. They include: (a) small datasets *Iono*, *Cancer*, *Pima*, *Sonar*, *Heart*, *Vote*, *WDBC*, *WPBC*, (b) medium dataset *Mushroom*, and (c) comparatively larger datasets *Adult* and *CodRna*.

Classification accuracy and kernel scalability results have been presented on small datasets (with many kernels). Scalability results (with 12 kernels due to memory constraints) have been provided for medium and large datasets. Due to space limitations, only a subset of our results on small datasets are presented. In addition, for ease of readability, we compare a smaller set of MKL methods (UNIFORM, GROUPEMKL and MWUMKL) in the main part of the paper. These MKL methods are the fastest among all baselines, as can be seen in the detailed comparisons presented in the appendices for all baselines on all datasets. For data scalability results, we first compare all baselines (see Figure 12 of Appendix A) on moderate size dataset *Mushroom*. For larger datasets *Adult* and *CodRna*, we compare MWUMKL only with UNIFORM which is the fastest among all baselines (as we show for *Mushroom*). In all cases, we omit the results for SILP [20], which takes significantly longer to complete (e.g. on *Sonar*, the smallest dataset in our pool, each iteration of SILP takes about 4500 seconds on average, whereas UNIFORM requires 0.03 seconds on average).

Similar to [17, 25], we test our algorithms on a base kernel family of 3 polynomial kernels (of degree 1 to 3) and 9 gaussian kernels (of width $\{2^{-3}, 2^{-2}, \dots, 2^5\}^2$). For small datasets, we use kernels constructed from individual features (i.e., a kernel is constructed by considering only one feature in the computation, setting all other features to zero) as well as from all features taken together. For medium and large datasets, due to memory constraints, we test only on 12 kernels constructed using all features. All kernels are normalized to trace m (the total number of kernels). For all experiments we report results averaged over 30 iterations (with standard deviations where applicable). In each iteration, 80% of the examples are randomly selected as the training data and the remaining 20% are used as test data. Feature values of all datasets have been scaled to $[0, 1]$. SVM regularization parameter C is 100 and is chosen by cross-validation. The duality gap (required as a stopping criterion for many baselines) as well as other settings are same as defined in the SimpleMKL toolbox [17]. For MWUMKL, we set $\varepsilon = 0.2$. In theory, MWUMKL requires $\tau = (8\rho^2/\varepsilon^2) \ln(n)$ number of iterations. However, we observed that in practice MWUMKL converges in much smaller number of iterations and we have found $\tau/32$ iterations to be sufficient for good performance on all of our datasets. Both ε and the iteration factor were chosen by cross-validation. Contrary to existing

²Technically since we are normalizing the range of each feature, we should use a smaller bandwidth set for each kernel; in future work we will release results with a bandwidth range of $\{2^{-8}, 2^{-7}, \dots, 2^0\}$.

works we do not compare the number of SVM calls (as MWUMKL does not explicitly use an underlying SVM) and the number of kernels selected.

Experiments were performed on a cluster of 30 machines with two different configurations: (a) 10 quad-core Intel Xeon E5405 CPUs with 32GB of RAM, and (b) 20 single-core Intel Xeon CPUs with 8GB of RAM. All methods have an outer test harness written in MATLAB. The methods we compare against are implemented using libSVM [5] based SVM solvers and MOSEK (<http://www.mosek.com>) based convex solvers (we note that these solvers are all implemented and optimized in C). MWUMKL also uses a test harness in MATLAB with an inner core written in C.

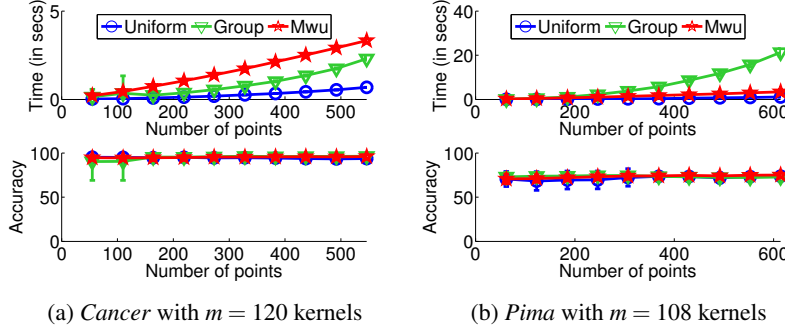


Figure 1: *Cancer* ($n = 683$, $d = 9$) and *Pima* ($n = 768$, $d = 8$).

two largest among the small datasets, namely, *Cancer* (Figure 1a) and *Pima* (Figure 1b). In each case, we report the number of kernels used. For both datasets, MWUMKL yields good accuracies. Additional detailed results (Figures 4b-11b in Appendix A) show that the accuracy of MWUMKL is either the best or close to it. The computational benefits for MWUMKL is not noticeable for *Cancer* which has a small training time. For *Pima* MWUMKL is significantly faster than GROUPMKL and quite close to UNIFORM.

Data Scalability. We start with results for *Mushroom* ($n = 8124$, $d = 112$) with 12 kernels. *Mushroom* is linearly separable and as expected all methods in Figure 2a achieve 100% accuracy. However both MWUMKL and UNIFORM are much faster as compared with GROUPMKL. Detailed results in Figure 12 (of Appendix A) show that MWUMKL and UNIFORM are the fastest among all baselines and are significantly faster. Hence, for the remaining experiments on large datasets, we only compare UNIFORM with MWUMKL.

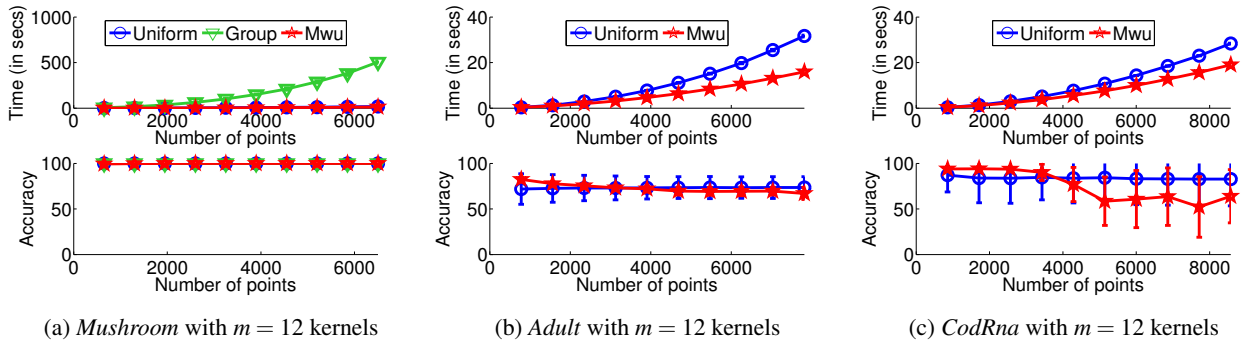


Figure 2: Results for *Mushroom* ($n = 8124$, $d = 112$), *Adult* ($n = 9768$, $d = 123$) and *CodRna* ($n = 9525$, $d = 8$).

Figures 2b and 2c present results for *Adult* ($n = 48842$, $d = 123$) and *CodRna* ($n = 59535$, $d = 8$), respectively. Note that although these datasets are large, with 12 kernels (on all features), we could only experiment on a subset of the entire data (9768 points for *Adult* and 9525 points for *CodRna*) due to memory constraints. On both datasets, MWUMKL is *much faster* when compared to UNIFORM and the trace lines indicate that the gap (between training times of UNIFORM and MWUMKL) will only get wider with further increase in data size.

For *Adult*, the accuracy of MWUMKL is comparable to UNIFORM. For *CodRna*, the accuracy of MWUMKL is better than UNIFORM for smaller data sizes but is slightly worse (though within error bounds) for larger data sizes.

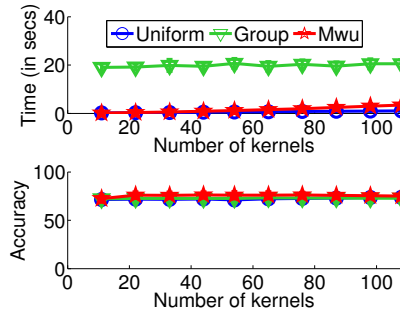


Figure 3: Time and Accuracy versus #kernels for *Pima*.

Kernel Scalability. We compared the scalability of MWUMKL with increase in number of kernels on small datasets. In Figure 3, we present kernel scalability results for *Pima* (the largest among the small size datasets). As can be seen, MWUMKL is among the fastest while yielding the best accuracy. Kernel scalability results on additional datasets have in provided in Appendix B (Figures 13-20). Excluding *Heart*, in all other cases, the accuracy of MWUMKL is either the best or among the best. We also observed that changing C from 100 to 10 makes MWUMKL achieve the best accuracies for *Heart*. The time taken by MWUMKL on small datasets is comparable to UNIFORM and GROUPEMKL. We believe that the benefits of MWUMKL (in terms of running time) with large number of kernels would be perceivable on bigger datasets (as is true for *Pima* in Figure 3 and Figure 15).

5 Discussions

Our results show that MWUMKL, while closely matching the accuracy of prior methods, runs significantly faster than these approaches, and is even faster than the simple UNIFORM heuristic for larger datasets. We believe that this substantially advances the state-of-the-art for fast MKL methods. Note that, with precomputed kernels, MWUMKL allows us to get to the largest datasets possible before memory limitations show up: this is clearly the next bottleneck in the quest for MKL methods that truly scale. One idea that we expect to be helpful is to make use of dynamic kernels, or even redesign the MWU scheme to explicitly only deal with rows of the kernel matrices instead of the entire matrix. In this regard, SMOMKL [23] presents interesting results on the large datasets we used, although they present no accuracy numbers, and their stated running times appear to be significantly larger (we were unable to complete a detailed comparison using our computational platform).

References

- [1] E. D. Andersen and K. D. Andersen. *The MOSEK interior point optimization for linear programming: an implementation of the homogeneous algorithm*, pages 197–232. Kluwer Academic Publishers, 1999.
- [2] A. Argyriou, R. Hauser, C. A. Micchelli, and M. Pontil. A DC-programming algorithm for kernel selection. In *ICML*, Pittsburgh, Pennsylvania, 2006.
- [3] S. Arora and S. Kale. A combinatorial, primal-dual approach to semidefinite programs. In *STOC*, San Diego, California, USA, 2007.
- [4] F. R. Bach, G. R. G. Lanckriet, and M. I. Jordan. Multiple kernel learning, conic duality, and the smo algorithm. In *ICML*, Banff, Canada, 2004.
- [5] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM TIST*, 2(3), 2011.
- [6] C. Cortes. Invited talk: Can learning kernels help performance? In *ICML*, Montreal, Canada, 2009.
- [7] C. Cortes, M. Mohri, and A. Rostamizadeh. Learning non-linear combinations of kernels. In *NIPS*, Vancouver, Canada, 2009.
- [8] C. Cortes, M. Mohri, and A. Rostamizadeh. Two-stage learning kernel algorithms. In *ICML*, Haifa, Israel, 2010.
- [9] N. Cristianini, J. Shawe-Taylor, A. Elisseeff, and J. S. Kandola. On kernel-target alignment. In *NIPS*, Vancouver, Canada, 2001.
- [10] M. Kloft, U. Brefeld, S. Sonnenburg, P. Laskov, K.-R. Müller, and A. Zien. Efficient and accurate lp-norm multiple kernel learning. In *NIPS*, Vancouver, Canada, 2009.
- [11] M. Kloft, U. Brefeld, S. Sonnenburg, and A. Zien. l_p -norm multiple kernel learning. *JMLR*, 12:953–997, 2011.
- [12] G. R. G. Lanckriet, N. Cristianini, P. Bartlett, L. E. Ghaoui, and M. I. Jordan. Learning the kernel matrix with semidefinite programming. *JMLR*, 5:27–72, December 2004.
- [13] C. A. Micchelli and M. Pontil. Learning the kernel function via regularization. *JMLR*, 6:1099–1125, December 2005.
- [14] C. S. Ong, A. J. Smola, and R. C. Williamson. Learning the kernel with hyperkernels. *JMLR*, 6:1043–1071, 2005.
- [15] F. Orabona and J. Luo. Ultra-fast optimization algorithm for sparse multi kernel learning. In *ICML*, Bellevue, USA, 2011.
- [16] P. Pavlidis, J. Cai, J. Weston, and W. N. Grundy. Wn: Gene functional classification from heterogeneous data. In *Proceedings of the Fifth Annual International Conference on Computational Biology*, 2001.
- [17] A. Rakotomamonjy, F. Bach, S. Canu, and Y. Grandvalet. More efficiency in multiple kernel learning. In *ICML*, Corvallis, USA, 2007.
- [18] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0262194759.
- [19] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.
- [20] S. Sonnenburg, G. Rätsch, C. Schäfer, and B. Schölkopf. Large scale multiple kernel learning. *JMLR*, 7:1531–1565, December 2006.

- [21] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999.
- [22] M. Varma and B. R. Babu. More generality in efficient multiple kernel learning. In *ICML*, Montreal, Canada, 2009.
- [23] S. V. N. Vishwanathan, Z. Sun, N. Ampornpunt, and M. Varma. Multiple kernel learning and the SMO algorithm. In *NIPS*, Vancouver, Canada, 2010.
- [24] Z. Xu, R. Jin, I. King, and M. R. Lyu. An extended level method for efficient multiple kernel learning. In *NIPS*, Vancouver, Canada, 2008.
- [25] Z. Xu, R. Jin, H. Yang, I. King, and M. R. Lyu. Simple and efficient multiple kernel learning by group lasso. In *ICML*, Haifa, Israel, 2010.
- [26] J. Ye, J. Chen, and S. Ji. Discriminant kernel and regularization parameter learning via semidefinite programming. In *ICML*, Corvalis, Oregon, 2007.
- [27] A. Zien and C. S. Ong. Multiclass multiple kernel learning. In *ICML*, Corvalis, Oregon, 2007.

A Accuracy Results on additional datasets

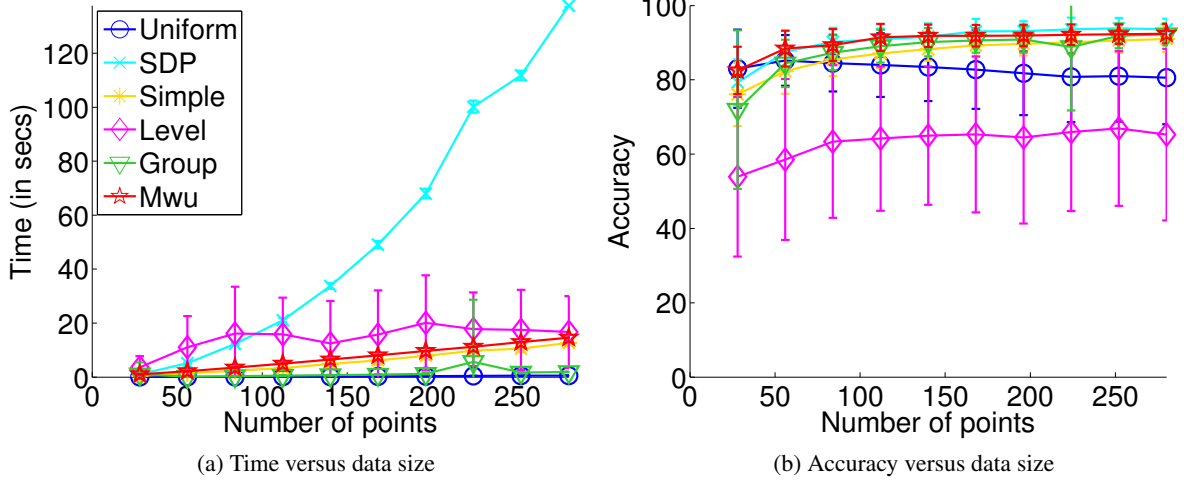


Figure 4: *Iono* ($n = 351, d = 33$) with $m = 12(33 + 1) = 408$ kernels.

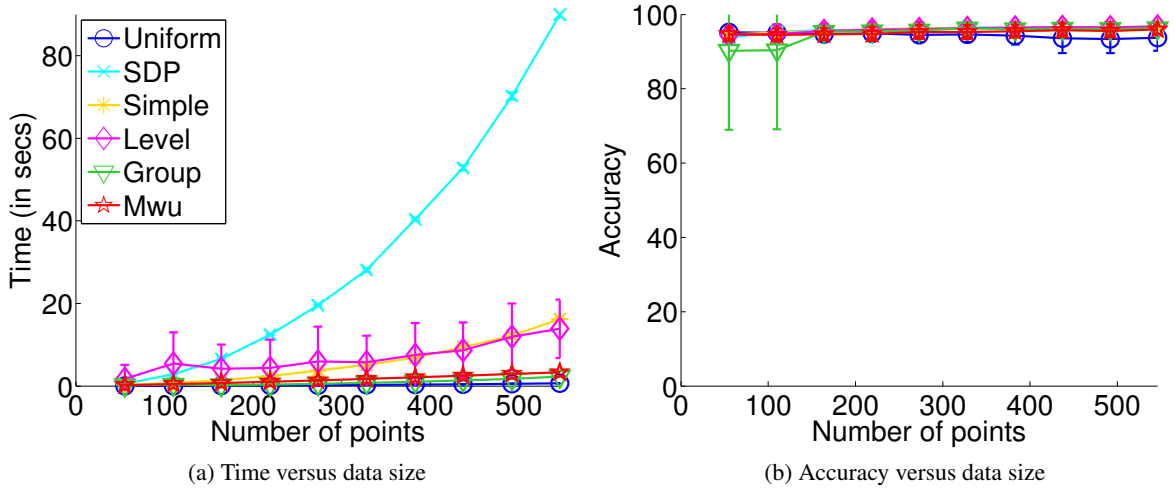


Figure 5: *Cancer* ($n = 683, d = 9$) with $m = 12(9 + 1) = 120$ kernels.

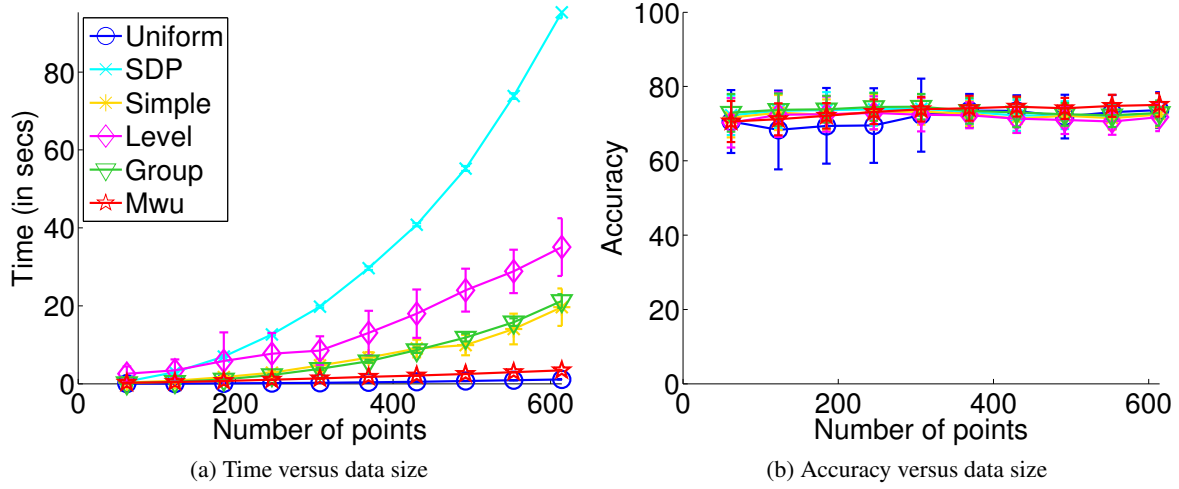


Figure 6: *Pima* ($n = 768, d = 8$) with $m = 12(8 + 1) = 108$ kernels.

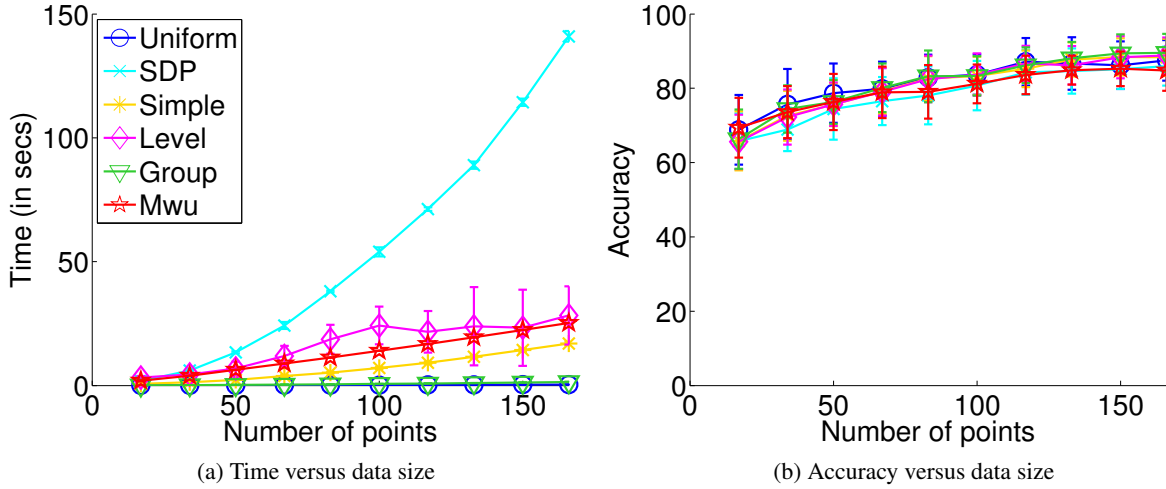


Figure 7: *Sonar* ($n = 208, d = 60$) with $m = 12(60 + 1) = 732$ kernels.

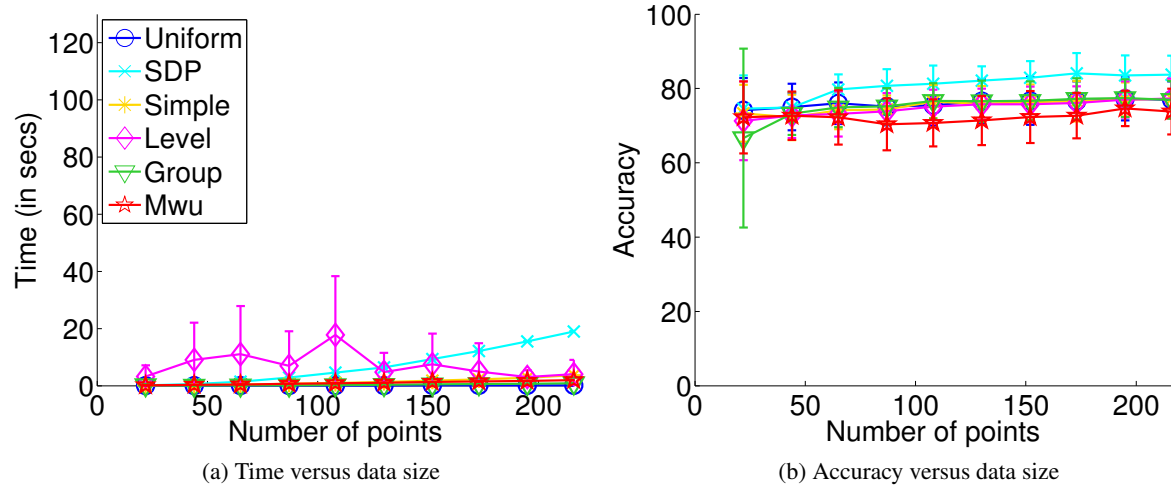


Figure 8: *Heart* ($n = 270, d = 13$) with $m = 12(13 + 1) = 168$ kernels.

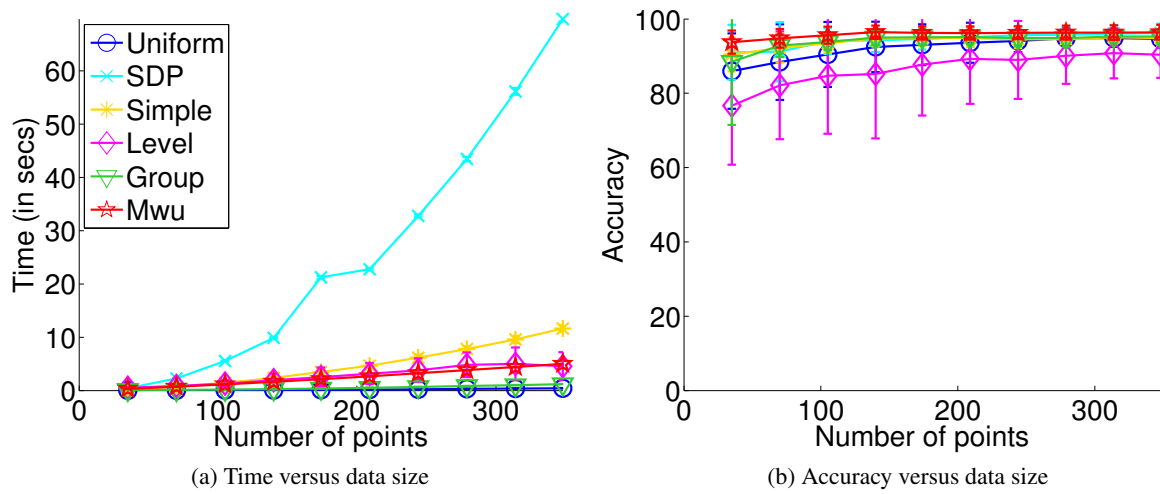


Figure 9: *Vote* ($n = 435, d = 16$) with $m = 12(16 + 1) = 204$ kernels.

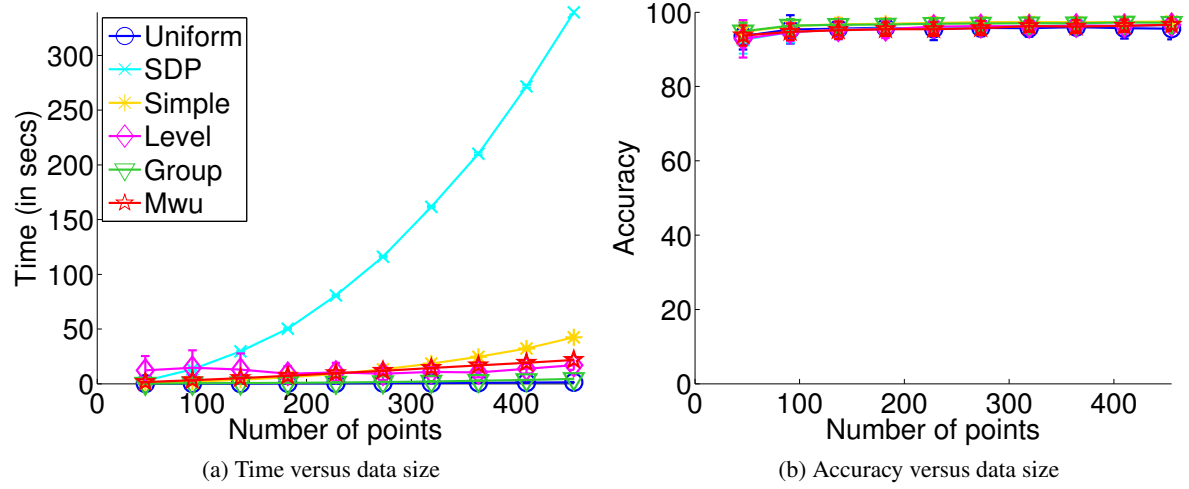


Figure 10: $WDBC$ ($n = 569, d = 30$) with $m = 12(30 + 1) = 372$ kernels.

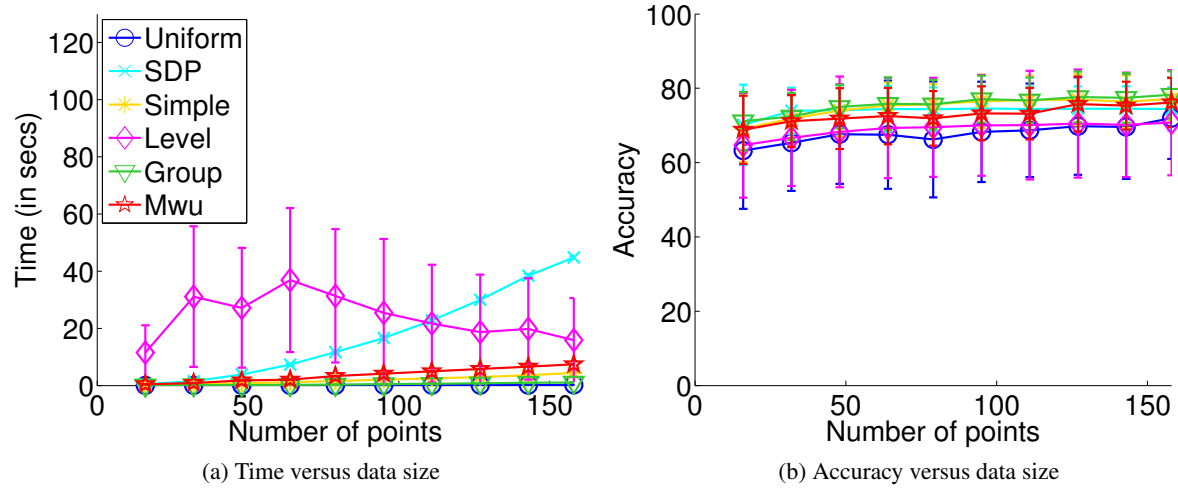


Figure 11: $WPBC$ ($n = 198, d = 33$) with $m = 12(33 + 1) = 408$ kernels.

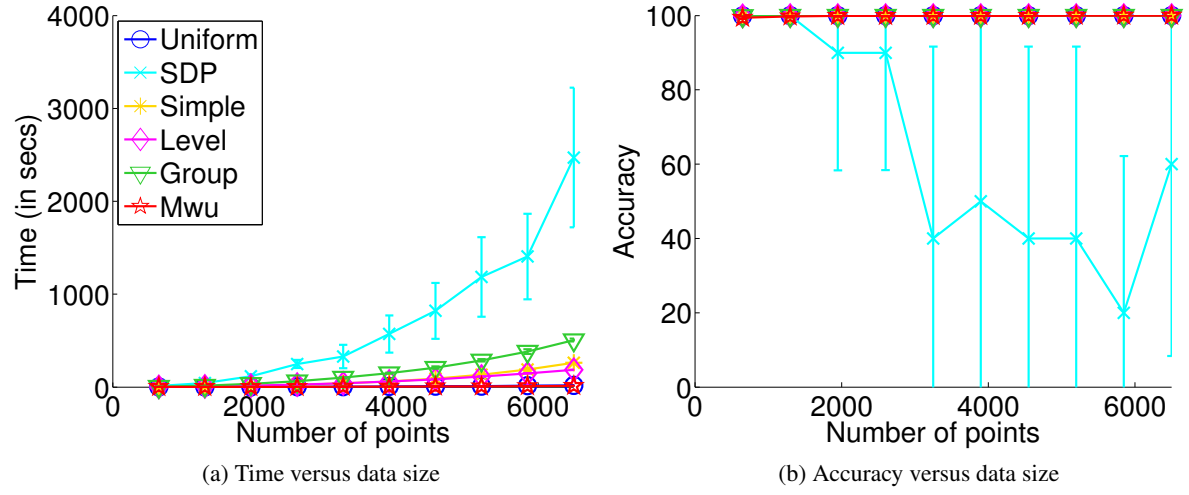


Figure 12: *Mushroom* ($n = 8124, d = 112$) with $m = 12$ kernels.

B Kernel Scalability Results on additional datasets

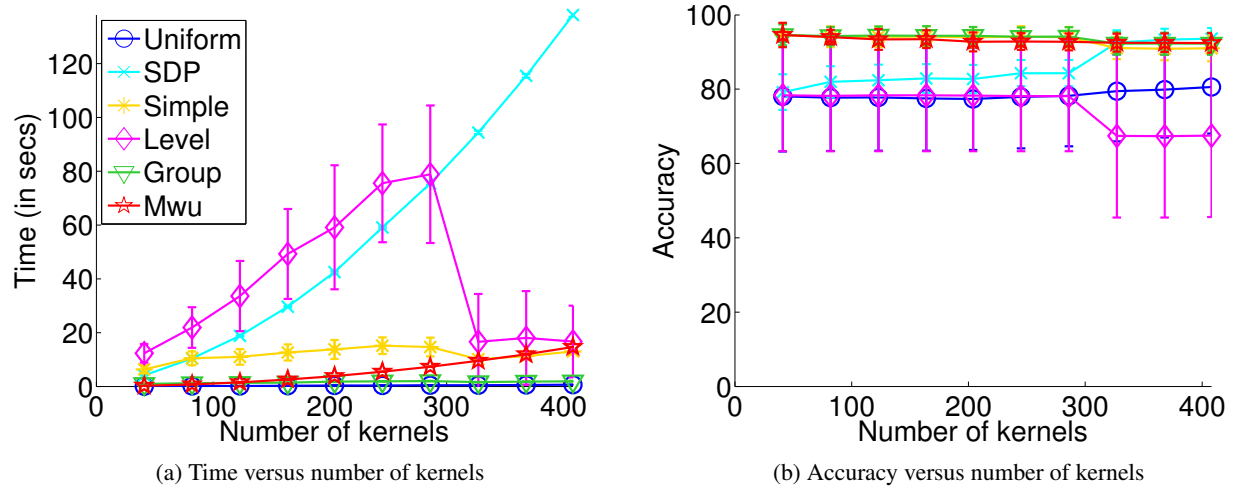


Figure 13: *Iono* ($n = 351, d = 33$) with $m = 12(33 + 1) = 408$ kernels.

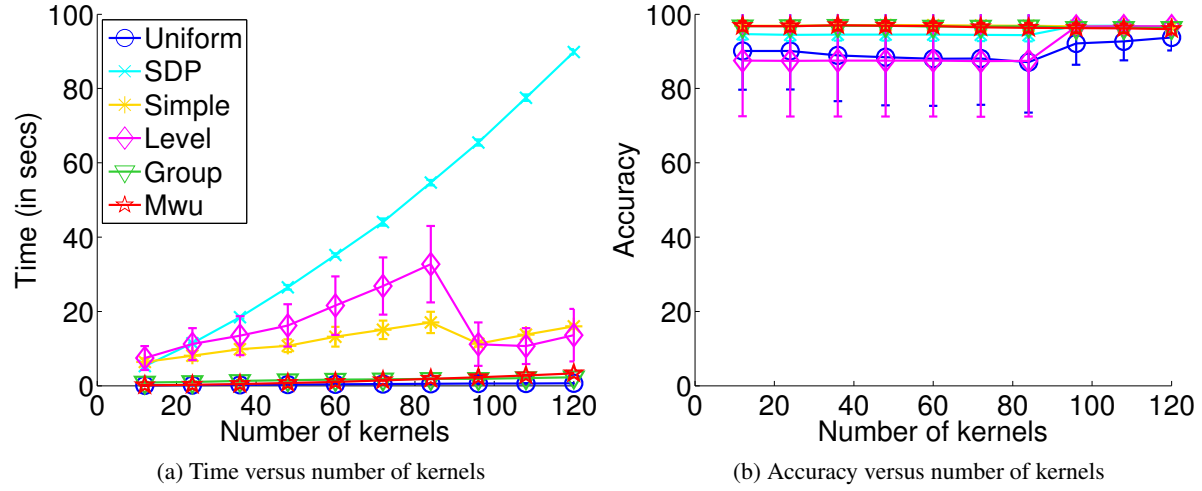


Figure 14: *Cancer* ($n = 683, d = 9$) with $m = 12(9 + 1) = 120$ kernels.

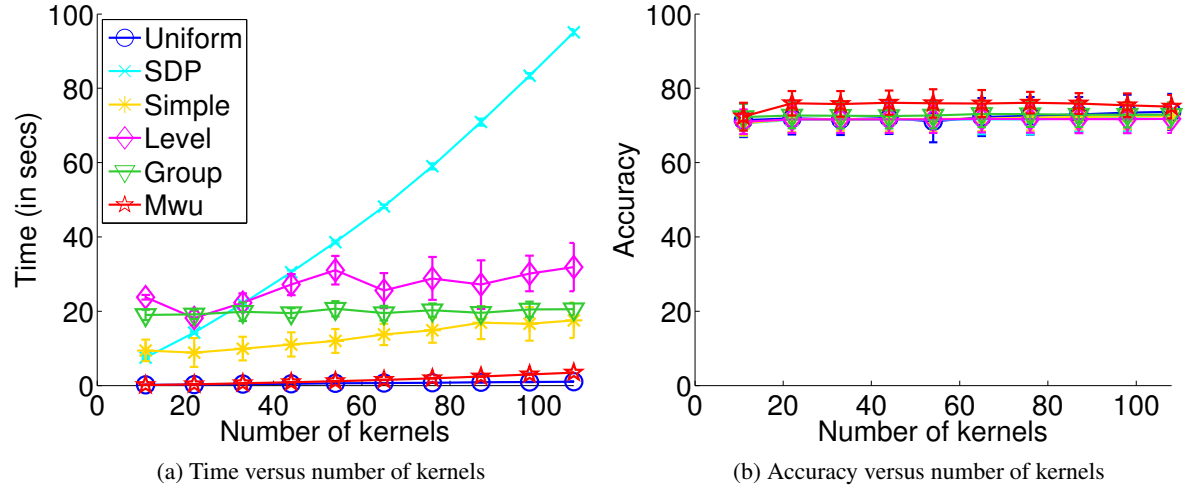


Figure 15: *Pima* ($n = 768, d = 8$) with $m = 12(8 + 1) = 108$ kernels.

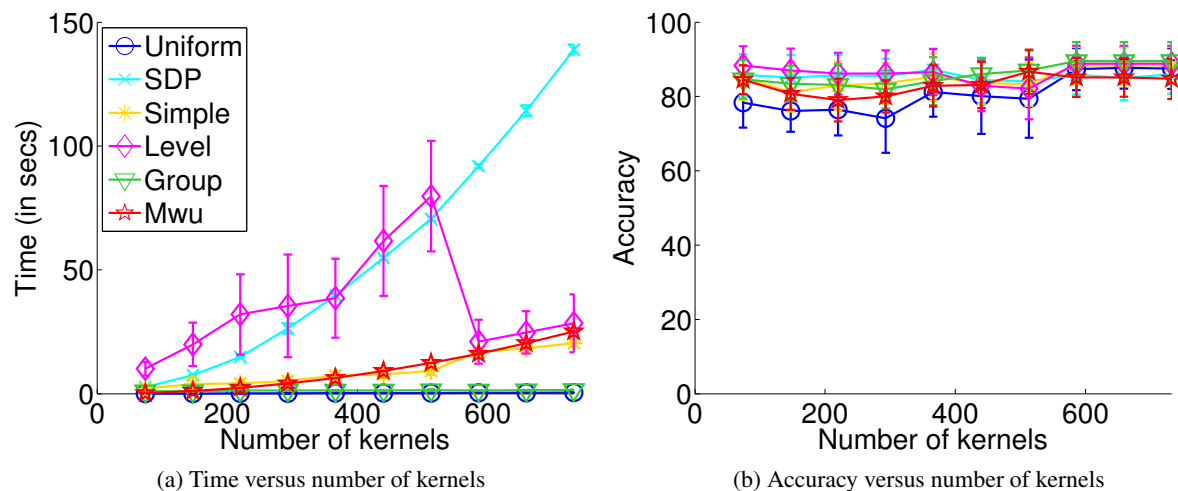


Figure 16: *Sonar* ($n = 208, d = 60$) with $m = 12(60 + 1) = 732$ kernels.

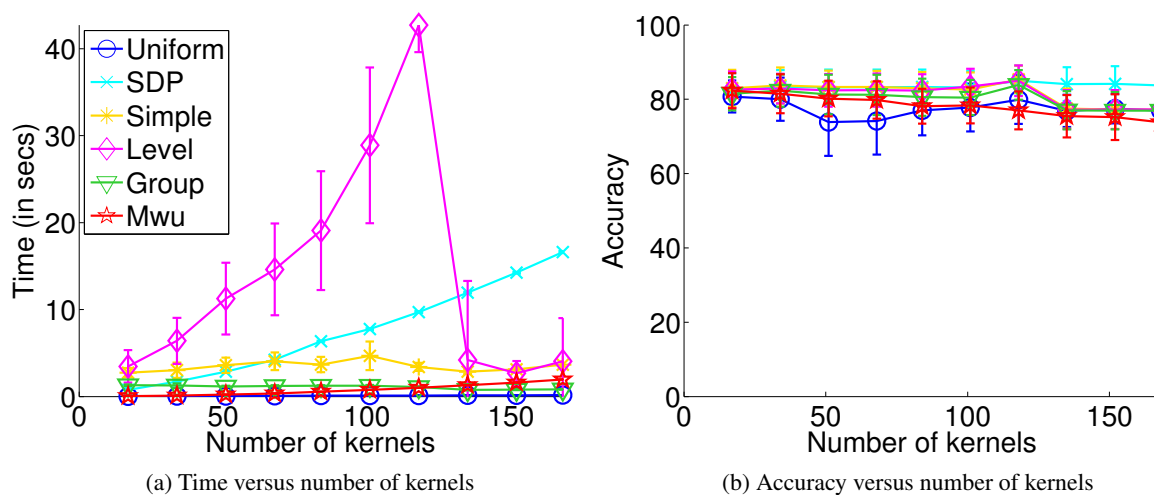


Figure 17: *Heart* ($n = 270, d = 13$) with $m = 12(13 + 1) = 168$ kernels.

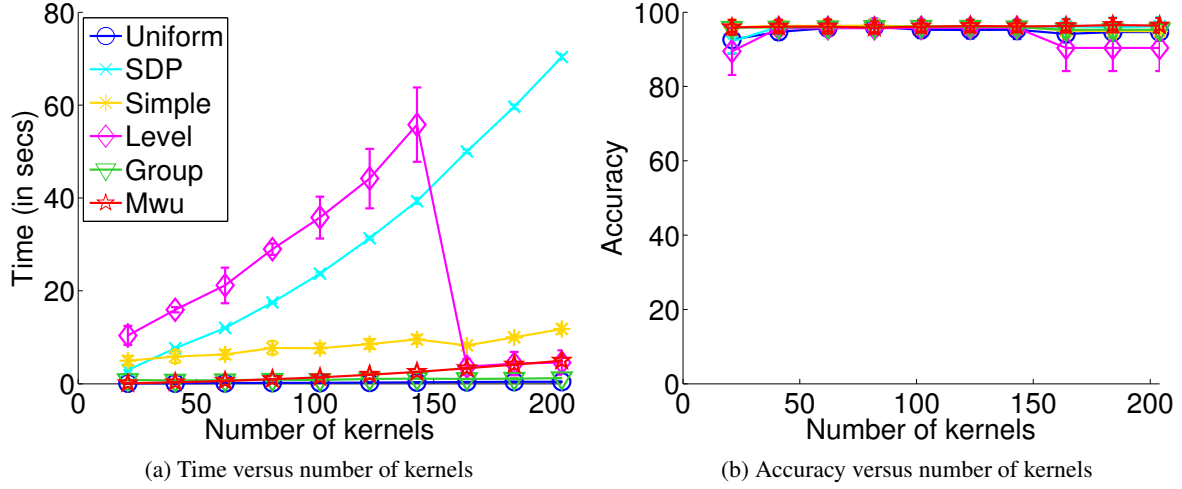


Figure 18: *Vote* ($n = 435, d = 16$) with $m = 12(16 + 1) = 204$ kernels.

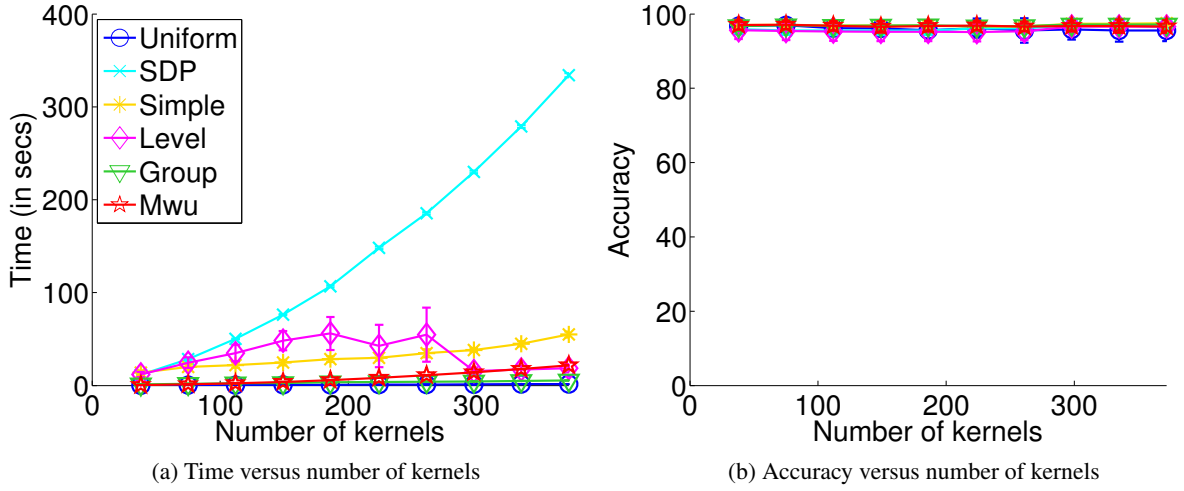
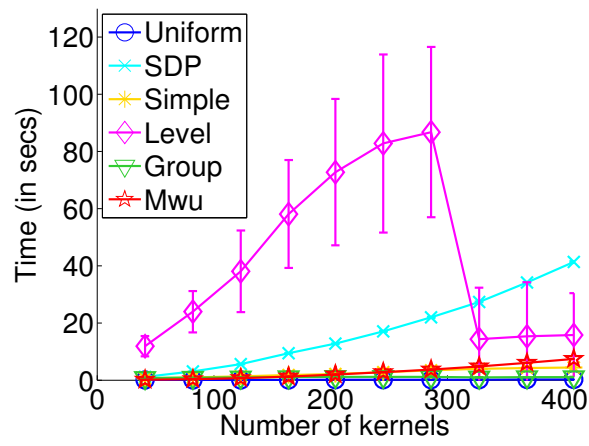
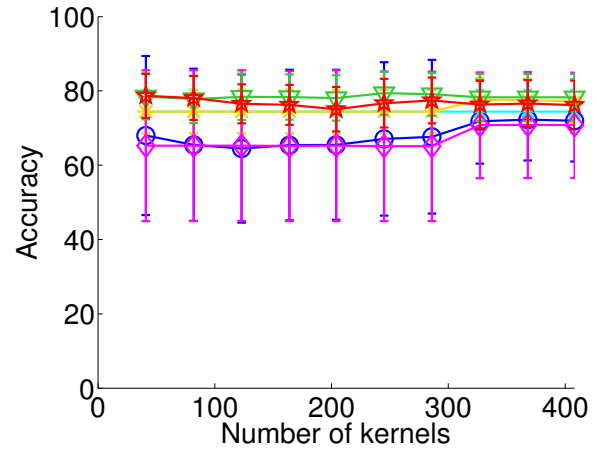


Figure 19: *WDBC* ($n = 569, d = 30$) with $m = 12(30 + 1) = 372$ kernels.



(a) Time versus number of kernels



(b) Accuracy versus number of kernels

Figure 20: $WPBC$ ($n = 198, d = 33$) with $m = 12(33 + 1) = 408$ kernels.